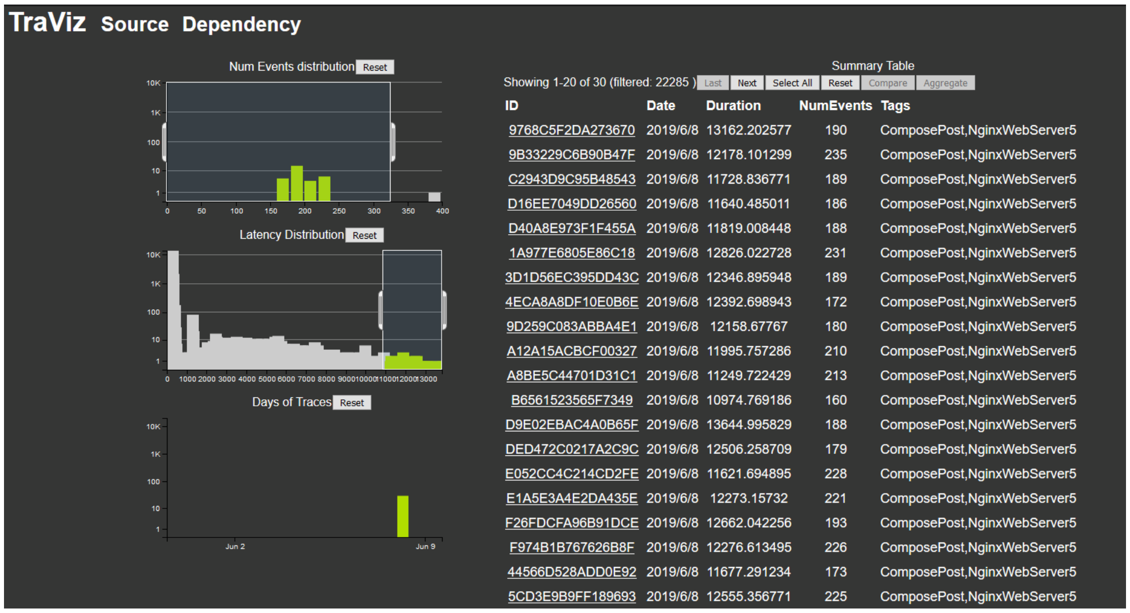
TraViz: Visualization of Traces in Distributed Systems

Vaastav Anand and Matheus Stolet



Summary Table

Showing 1-20 of 30 (filtered: 22285)

ID	Date	Duration	NumEvents	Tags
9768C5F2DA273670	2019/6/8	13162.202577	190	ComposePost,NginxWebServer5
9B33229C6B90B47F	2019/6/8	12178.101299	235	ComposePost,NginxWebServer5
C2943D9C95B48543	2019/6/8	11728.836771	189	ComposePost,NginxWebServer5
D16EE7049DD26560	2019/6/8	11640.485011	186	ComposePost,NginxWebServer5
D40A8E973F1F455A	2019/6/8	11819.008448	188	ComposePost,NginxWebServer5
1A977E6805E86C18	2019/6/8	12826.022728	231	ComposePost,NginxWebServer5
3D1D56EC395DD43C	2019/6/8	12346.895948	189	ComposePost,NginxWebServer5
4ECA8A8DF10E0B6E	2019/6/8	12392.698943	172	ComposePost,NginxWebServer5
9D259C083ABBA4E1	2019/6/8	12158.67767	180	ComposePost,NginxWebServer5
A12A15ACBCF00327	2019/6/8	11995.757286	210	ComposePost,NginxWebServer5
A8BE5C44701D31C1	2019/6/8	11249.722429	213	ComposePost,NginxWebServer5
B6561523565F7349	2019/6/8	10974.769186	160	ComposePost,NginxWebServer5
D9E02EBAC4A0B65F	2019/6/8	13644.995829	188	ComposePost,NginxWebServer5
DED472C0217A2C9C	2019/6/8	12506.258709	179	ComposePost,NginxWebServer5
E052CC4C214CD2FE	2019/6/8	11621.694895	228	ComposePost,NginxWebServer5
E1A5E3A4E2DA435E	2019/6/8	12273.15732	221	ComposePost,NginxWebServer5
F26FDCFA96B91DCE	2019/6/8	12662.042256	193	ComposePost,NginxWebServer5
F974B1B767626B8F	2019/6/8	12276.613495	226	ComposePost,NginxWebServer5
44566D528ADD0E92	2019/6/8	11677.291234	173	ComposePost,NginxWebServer5
5CD3E9B9FF189693	2019/6/8	12555.356771	225	ComposePost,NginxWebServer5

Fig. 1. TraViz overview layout, showing a filtered selection of distributed traces using filters on distributions of multiple dimensions of the distributed traces dataset.

Abstract— In this work we present TraViz, an interactive visualization tool for exploring and analysing distributed traces to troubleshoot and debug performance problems in distributed systems. Through the use of composite filtering of multiple dimensions of the dataset, TraViz provides developers an easy to use way to find outlier traces. With TraViz's traceview swimlane idiom, the users can drill down into a single trace to analyze the trace for performance issues at the detail level of an operating system thread. TraViz's comparison idiom allows users to compare the Graphical structure of 2 traces of interest to highlight the key differences between the traces. The aggregation idiom allows users to find uncommon occuring events across traces by constructing a luminance coded super graph of all the distributed traces. Additionally, TraViz is the first visualization tool that provides an integrated view of the static source code of the system with dynamic information collected about the system via distributed traces which can help the users in identifying locations in the source code that would be ideal for performance optimizations.

Index Terms—Distributed Tracing, Graph Comaprison, Graph Aggregation, Source Code Integration

1 Introduction

Distributed systems are prevalent in society to the extent that billions of people either directly or indirectly depend on the correct functioning of a distributed system. From banking applications to social networks, from large-scale data analytics to online video streaming, from web searches to cryptocurrencies, most of the successful computing applications of today are powered by distributed systems. The meteoric rise of cloud computing in the past decade has only increased our dependence on these distributed systems in our lives.

Tasks like monitoring, root cause analysis, performance comprehension require techniques that cut across component, system, and machine boundaries to collect, correlate, and integrate data. Distributed Tracing is one such cross-cutting technique that correlates events across the system to a specific request by propagating a unique context per system with the request. A trace represents the path of one request through the system and contains information such as the timing of requests, the events executed, and the nodes where these events were executed. Moreover, traces can be used to identify slow requests and understand the difference between request executions.

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

Although distributed traces carry vital information for debugging and for general system understanding, there have been question marks against the usability of distributed tracing [10, 6]. This has been primarily due to the lack of good analysis tools for analysing the datasets. Specifically, existing visualization tools don't provide an interface for exploring the dataset of traces and finding potential outliers. Additionally the tools don't have a way of showing structural differences between traces and structural similarities across a group of traces. Distributed Traces are such a rich source of data that can be useful for debugging and even resource allocation but existing tools have limited the usability of distributed tracing as these tools have barely scratched the surface of the plethora of analysis tasks that can be carried with the data available in distributed traces.

To rectify the shortcomings of existing analysis tools and to make distributed tracing more useful, in this paper, we present a new visualization tool called TraViz to analyze and explore datasets of distributed traces. TraViz exposes an exploration dashboard which allows the users to find outlier traces by filtering across distributions of multiple dimensions of the trace dataset. TraViz also integrates the performance information with the source code information of the distributed system, highlighting files and lines in the source code that appear the most across traces. TraViz also provides idioms for comparing two different traces as well as aggregating multiple traces into a single super-trace. To provide the users with a sense of familiarity, TraViz also provides popular single-trace visualizations that are prevalent in state-of-the-art distributed tracing visualization tools. With the use of a tiny informal user study, we show that TraViz achieves our goals of improving the usability of distributed tracing through enriching the analytical power of the users tasked with analyzing the distributed trace datasets.

2 Related Work

Jaeger [1] is an open-source distributed tracing project that provides libraries for instrumenting distributed systems in different languages as well as a frontend for viewing the traces produced from these systems. Jaeger has vis idioms for visualizing a single trace, for visualizing the dependencies between services of the system, as well as an idiom for comparing 2 different traces. However, it doesn't have any idiom for viewing an aggregate form of multiple traces or for comparing groups of traces.

LightStep [2] is a start-up company that creates solutions for real-time tracking of requests and metrics in large-scale distributed systems. LightStep has vis idioms for visualizing the structure of a single trace, for visualizing the critical path of a single trace, and for visualizing various metrics collected for multiple traces. Like Jaeger, LightStep also lacks a viz idiom for viewing aggregate traces and for comparing groups of traces.

ShiViz [3] is an interactive visualization tool that visualizes communication graphs from distributed system execution logs. As ShiViz is designed for visualizing logs and not individual traces, ShiViz does not support comparisons of multiple traces.

The current state-of-the-art visualization tools for distributed tracing have historically focused more on visualizing the structure and performance characteristics of the traces using a waterfall idiom where each process in the trace is modeled as a separate level in the waterfall. However, these tools don't provide a idiom for finding outlier traces and have very rudimentary visualizations for comparing different traces which are tasks that target users actively wish to accomplish. With TraViz, we address some of the shortcomings of existing tools by providing a way for the users to compare the structure of two traces, aggregate a group of traces, and integrate the traces with source code to provide more context for debugging. We also provide an outlier exploration dashboard that allows the users to filter along multiple dimensions of the dataset to quickly explore the dataset and find possible outlier and erroneous traces.

3 Data Abstraction

We have 2 datasets that we will be visualizing using TraViz. The first dataset is called HDFS and contains the **collection of traces** and **collection of processes** from a Hadoop file system used for distributed storage and big data processing and the **corresponding source code**. The second dataset is called socialNetwork and was obtained from the DeathStarBench open-source benchmark for cloud microservices [5]. It contains the **collection of traces**, **collection of processes** as well as the corresponding **source code**. Each distributed system comprises of multiple services that run on machines as different processes. The HDFS dataset has a total of 71,001 traces and 18 processes. The socialNetwork dataset has a total of 22,286 traces and 20 processes.

We discuss the attributes of each **trace**, **process**, and **source code** in detail below.

3.1 Process

The process entity represents a service running in the distributed system. The attributes for Process are shown in Table 3.1. In addition to the attributes listed, each process also comprises of multiple threads which corresponds to an OS or Language Runtime level thread.

3.1.1 Thread

Each thread has a categorical attribute **tid**, which is a unique identifier for a thread within a process.

3.2 Trace

A trace represents a request from a client to a service and shows the path of the request through the distributed system. A trace has five attributes: *id*, *duration*, *start_stamp*, *list_of_tags*, and *list_of_events*.

- *id* of the trace is a categorical attribute used to identify a trace.
- *duration* of the trace is the total time taken to service a request.
- *start_stamp* is the unix timestamp of the time when the trace was started.
- *list_of_tags* is a list of human defined keywords that serve as the metadata for the trace. There are on average two tags per trace in both datasets, but the

Attribute	Type	Description	HDFS Range/Cardinality	socialNetwork Range/Cardinality
id	Categorical	Unique identifier of a process	18	20
name	Categorical	Name of the service the process is running	4	13
host	Categorical	Name of the machine on which the process is running	9	13

Table 1. Attributes of a Process

socialNetwork dataset has a total of 8 different tags and the HDFS dataset has a total of 22285 different tags.

- *list_of_events* attribute is a list of the events that happened in a trace. The events in the list are partially ordered [7], with events that caused another event preceding the caused event in the list. Such a partial causal relationship between the events forms a DAG.

For the socialNetwork dataset, the number of events per trace range from 3 to 398 with a mean of 99 events. For the HDFS dataset, the number of events per trace range from 0 to 9697 with a mean of 1427 events.

3.2.1 Event

Events are important things that happen in a system, such as acquiring a lock, sending a request to another server, or performing an update. These events are defined by the developer as instrumentation in the source code. Events can be considered as anything a developer thinks is useful enough to log. The attributes of an event are shown in Table 2.

4 Tasks

We have identified a total of six tasks that we want to support with our viz tool. These tasks are finding outliers and providing an overview of the dataset, integrating tracing data to the source code, analysing a single trace, understanding the dependencies between services in a distributed system, comparing two traces, and aggregating multiple traces. Each task is clarified through an example below.

Scenario 1: Overview and finding outliers

A developer wants to find requests that are slower than usual in a distributed system. This information should be consumed by the developer in a way that helps identify what traces are worthy of a detailed analysis. For example, identifying unsually slow requests is useful at selecting what traces should be inspected to understand the path of their requests.

Scenario 2: Integrating with the source code

A developer wants to locate files that are highly active. This information can be used to understand what areas of the source code are responsible for the activity in the system, and thus require more attention. These insights can be used to alert developers that new code to those files should be thoroughly tested and reviewed, since it belongs to a critical part of the system. The developer should be directed to the source code once a file of interest is identified.

Scenario 3: Analyzing one trace

A developer wants to have detailed information on a request. Tracing data can be consumed to reveal information such as the path of the request, the time the events in a request were triggered, and what threads executed a request. This information can reveal detailed system information that can be used to optimize the code.

Scenario 4: Finding service dependencies

A developer wants to find how many services communicate with a specific service. This task entails locating the node in a graph representing the service of interest, and following the links from that service to other services. This information can be used to identify the services in a microservice that are dependent in many other services, which can be useful in helping developers find the services that should be refactored to minimize dependencies.

Scenario 5: Comparing traces

A developer wants to compare two requests to understand why one is faster than the other. This task takes the shape of deriving metrics to highlight similarities and differences between the events in both traces. This information can then be presented to the user in a way that the differneces are discovered, while still maintaining enough contextual information from both traces to understand the path of the request.

Scenario 6: Aggregating traces

A developer wants to summarize the information on multiple traces with the intent of revealing trends in the data. This task takes the form of aggreagating similar traces into a graph, so the topology of these aggregated traces can be visualized. Aggreagating traces is a useful task for distributed systems developers because it helps condense the information of multiple requests into a summarized format that can be consumed for analysis.

5 Implementation Approach

The Implementation of our solution consists of 3 parts: (i) MySQL database, (ii) Backend REST API server implemented in Go, and (iii) Frontend React WebApp that implements our visualization idioms.

The MySQL database contains 4 different tables. The 4 tables are the overview, sourcecode, tags, and dependencies table. These tables contain values for various derived attributes that we make use of in our idioms. The database setup code is 39 lines of MySQL.

The Go Backend is responsible for processing the raw JSON format of distributed traces and populating the tables created in the MySQL database. The backend also hosts a REST API server to service requests from the frontend which queries over the database to obtain information about the traces. The backend implementation also contains a minimal graph kernel library for computing the similarities between 2 graphs. Specifically, we implemented the Weisfeiler-Lehman graph kernel [9]. The Go Backend, including the graph kernel library, is implemented in 1600 lines of code.

The frontend is a React web application where we implement our visualization idioms. We use d3 to implement our viz idioms. We make use of the dc and crossfilter libraries to implement our overview and source code dashboard to provide linked views. We use the react-d3-

Attribute	Type	Description
id	Categorical	Unique identifier of an event
trace_id	Categorical	Unique identifier of the trace to which the event belongs
process_id	Categorical	Unique identifier of the process on which the event occured
thread_id	Categorical	Identifier of the thread in a process on which the event occured
hrt	Quantitative	High Resolution (ns) Unix Timestamp
label	Free-Form Text	Developer-added annotation for the event
full_path	Categorical	Full path of the file in the source code where the event was logged.
source_line	Categorical	Line in the source code where the event was logged. Takes the format of Full_path:Line_number.

Table 2. Attributes of an event

graph library to implement our graph-based visualization idioms. Our swimlane visualization idiom for showing a detailed view of a single distributed trace is based on the visualization idiom used by X-Trace server [4, 8] as we believe it is very effective and very useful in providing detailed performance hints to the developers. Our frontend implementation is 3000 lines of javascript and css on top of the libraries we used.

6 Solution

Traviz accomplishes six tasks. These tasks are finding outliers and overviewing the dataset, integrating source code and tracing data, extracting a detailed view of an individual trace, analyzing the dependencies between the services in a distrbuted system, comparing two traces, and aggregating multiple traces into a single visualization. Our solution to each one of these tasks is described in the subsections below.

6.1 Outliers and Overview

For this task we consume a set of traces and identify outliers and patterns. We provide a visual representation of the distribution of the number of events in a trace, trace duration, and date of a trace. We achieve this by using crossfilter to display the distributions in a bar chart and arranging the tracing data in a table.

To reduce the cognitive load on the user, we reduce the number of items on display by using crossfilter to select areas in the distribution charts, which causes the data table to be filtered accordingly. For example, if a user selects the area between 1000ms and 2000ms in the latency distribution chart, the table will only display traces that have a duration between 1000ms and 2000ms. The table can also be sorted to help users quickly find traces. Visually representing the distributions in bar charts and filtering the data from these distributions allows users to swiftly find outliers. For example, traces with more events than usual can be quickly identified from the bar chart and selected using our filtering functionality.

Outliers and Overview
What: data - A collection of traces
Why: tasks - Find outliers - Identify patterns
How: reduce - Filter items using attributes such as number of events, duration, and date of a trace. **How: show** - Display traces on table that can be sorted and filtered.

6.2 Source Code Integration

Our source code integration tool consumes traces and derives the number of events triggered by a line in the source code. This tool is useful at identifying what files - and lines in the source code of that file - produce the most events. Developers can use the source code integration tool to identify what areas of their code are heavily utilized.

Our source code integration tool uses two bar charts to encode the relevant information. One of the charts aggregates the total number of events in a file and displays one bar per file. If a user clicks on the bar for one of the files, Traviz displays another bar chart adjacent to it that reveals the number of events in each source code line of the file. In both charts, we encode the number of events with the lenth and luminance of the bar, where higher luminance means that more events spawned from that file or source code line. To complete the source code integration, Traviz allows users to click on a bar in the source code line chart. A mouse click causes the application to redirect the user to the line in a Github repo hosting the project being visualized.

Source Code Integration
What: data - A collection of traces
Why: tasks - Identify what files produce events - Identify what lines in a file produce the most or least events
How: display - Display number of events in a file and number of events originating from a line in the source code using bar charts **How : encode** - Encode number of events in a file and number of events originating from a line in the source code using the size of the bar. - Encode number of events in a file and number of events originating from a line in the source code using the luminance of the bar.

6.3 Individual Trace Analysis

The individual trace analysis tool complements the overview tool and allows the details of a specific trace to be analysed. This tool displays the timestamp of the events in a trace, the thread where an event was executed, and the events that originated events in other threads. This tool is useful for a distributed systems developer because it allows a trace to be dissected, so that the path of a request can be observed. The individual trace analysis tool encodes each thread as a a lane, the time of an event as the position on the x-axis, and the thread of an event as the position on the y-axis. Furtermore, if an event triggered an event in another thread, we show this relationship with a connecting line.

Individual Trace Analysis
What: data - One trace
Why: tasks - Observe the timing of events in a trace - Identify events that triggered events in other threads
How: encode - Encode each thread as a lane - Encode the time of an event as the position of a point on the x-axis - Encode the thread of an event as the position of a point on the y-axis - Identify events that triggered events in other threads with a connecting line

6.4 Service Dependency Analysis

The service dependency analysis derives the total number of messages issued by a service. This tool is important for comprehending distributed systems because it allows developers to understand the dependency relationship between services. For example, in a micro-service architecture, the dependency graph helps understand what services talk to each other and what services talk to the most services. To visualize the dependencies, we arranged the services into a node-link graph, where each service is a node, and the dependency is a link between the nodes. We also encoded the degree of each node as the area of a circle, so that services that talk to many services can be easily recognized.

Service Dependency Analysis
What: derived data - Total messages issued by a service
Why: task - Understand the dependency between services
How: arrange - Arrange services into a node-link graph - Services are represented as nodes - Dependencies between services are represented as links between nodes **How: encode** - Encode the amount of dependencies in a service with the area of the node

6.5 Trace Comparison

In this tool we take two traces and merge them in a way that the user can indentify the differences and similarities between the traces. Understanding the differences and similarities between a trace can help developers identify why some requests take longer than others.

To build this tool we assigned each event in both traces a group between one and three. The events are arranged into a node-link graph, where each event is a node and the link encodes the parent-child relationship between events. We encode group three nodes as squares and groups one and two as circles. We also encode the group a node belongs to using hue. Group three nodes are aggreagated together, so that the total number of nodes, and consequentially cognitive load, are reduced. Users can disaggregate group three nodes if they want to take a look at the full graph. If users want to look at the details of an event, they can click on a node, which causes the attributes of the selected event to be displayed on the side.

Trace Comparison
What: data - Two traces
Why: tasks - Identify differences between traces - Observe patterns
How: arrange - Arrange events into a node-link graph - Events are represented as nodes - Parent-child relattionships between events are represented as links between nodes **How: encode** - Encode nodes in group 3 as squares and nodes in groups 1 and 2 as circles - Encode the group of an event using hue **How: aggregate** - Aggregate group 3 nodes to reduce the size of the graph

6.6 Trace Aggregation

The trace aggregation tool helps users see the big picture. It consumes a collection of traces and arranges them into a node-link graph, where nodes represent events and links represent the parent-child relationships between the events. To condense the information from multiple traces into a more manageable graph, we aggregate the events from the same source code line into one node. We also visualize the number of events in a node by using luminance, where high luminance means a node is responsible for many events, and low luminance means that a node is responsible for few events. A node can be selected by clicking, which results in detailed information about that event - such as the event id, timestamp, and thread id - to be displayed. The aggreagation tool also allows the structure from multiple traces to be analysed, while reducing the amount of nodes on display.

Trace Aggregation
What: data - A collection of traces
Why: tasks - Visualizing the big picture - Analyzing multiple traces
How: arrange - Arrange avents into a node-link graph - Events are represented as nodes - Parent-child relattionships between events are represented as links between nodes **How: aggregate** - Aggregate events from the same source code line as one node **How: encode** - Encode number of events in a node with luminance - High luminance means a node is aggregating many events - Low luminance means a node is aggregating few events

7 Results

7.1 Use Case 1: Outlier detection

The user has received multiple bug reports from different customers about some requests taking tens of seconds to complete when they usually take fewer than a second to complete. To investigate, the user opens up TraViz and sees the distribution of the traces. The user can quickly see that there are a small number of traces that have indeed taken unusually long as compared to other traces. The user selects a specific duration range to filter so that they can only see the list of traces that have their duration within that range. To further narrow down, the user selects an events range to further reduce the number of traces as shown in Figure 5. The user can now start performing root cause analysis.

7.2 Use Case 2: Performance Analysis on a single trace

Now that the user has a picked a filtered set of traces, the user can take a look at an individual trace to figure out the root cause of the performance issue. The user clicks on one of the traces from the filtered list to see the detailed view of the trace **??**. From a cursory look, the user can identify which thread takes the longest time to complete its execution and correctly identify the set of events that caused delay on the trace.

7.3 Use Case 3: Finding differences in pair of traces

The user wants to compare the structure of two of the outlier traces from the filtered list in Figure 5 to see if the outlier traces are very different or not. The user selects two traces of their choice and presses the compare button to produce a comparison graph between the traces as shown in Figure 4. The comparison graph has a lot of nodes which suggests that the selected traces are quite different. The user can click on circular nodes to look at details of the event that the node corresponds to. The user is mostly confused with this visualization due to the hairball effect.

7.4 Use Case 4: Finding similarities in groups of traces

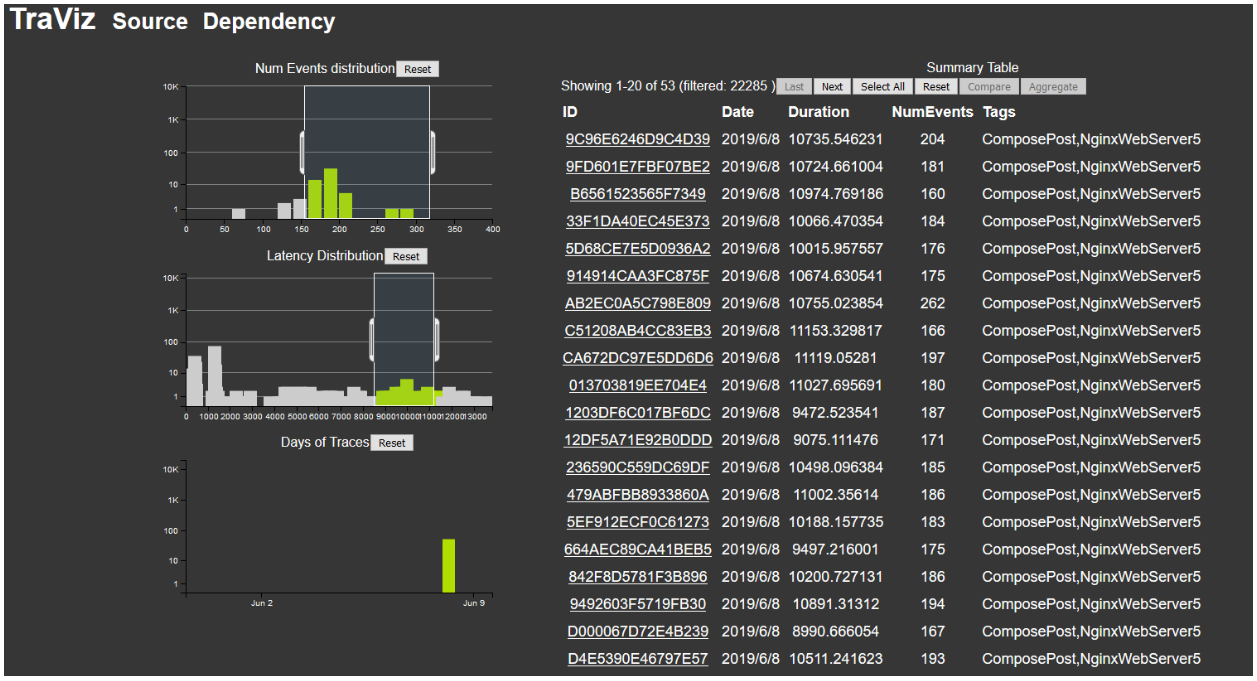
The user wants to aggregate some of the outlier traces from the filtered list in Figure 5 to generate a super graph that represents the structure of outlier traces. The user selects 20 traces using the select all button. The user then clicks the Aggregate button which produces the graph shown to the user in Figure **??**. The user notices that most nodes have bright luminance which suggests to the user that most of the nodes don't appear in a lot of traces. However, the user notices one dark node which the user clicks to see the details of the event which is common across a large proportion of the traces. Seeing the aggregate graph reminds the user of spaghetti and meatballs as the user is not able to fully understand what the aggregate graph is telling the user. The user wishes for a better graph layout in the next update of TraViz.

7.5 Use Case 5: Source Code Optimization

The user wants to know what files in the source code, Specifically what lines in the source code are producing the most number of events in traces as well as producing the least number of events in traces. The user clicks on the Source tab on the navigation bar to land at the source page to see the distribution of events across files in sorted order as shown in Figure 6. To view the distribution of events across the lines in a particular file, the user clicks on the bar with the name of the target file. This gives rise to another chart which shows the distribution of the events across lines in that file in sorted order as shown in Figure 7. To view the line in the context of the source code, the user clicks on the bar of that specific line to get redirected to that specific line in the specific file on github.

7.6 Use Case 6: Service Dependency Analysis

The user wants to know how the different services in the distributed system interacts with each other. Specifically, the user wishes to know the load each service is receiving from all the other services in the system. To accomplish this, the user clicks on the Dependency tab on the navigation bar to land at the dependency page to see the dependency graph as shown in Figure 8. To find out detailed information about each service, the user clicks on the node with the particular service to bring up the detailed view of the service which shows the number of messages received by that service from every other service, and the number of messages sent by that service to every other service.



Summary Table

Showing 1-20 of 53 (filtered: 22285) Last Next Select All Reset Compare Aggregate

ID	Date	Duration	NumEvents	Tags
9C96E6246D9C4D39	2019/6/8	10735.546231	204	ComposePost,NginxWebServer5
9FD601E7FBF07BE2	2019/6/8	10724.661004	181	ComposePost,NginxWebServer5
B6561523565F7349	2019/6/8	10974.769186	160	ComposePost,NginxWebServer5
33F1DA40EC45E373	2019/6/8	10066.470354	184	ComposePost,NginxWebServer5
5D68CE7E5D0936A2	2019/6/8	10015.957557	176	ComposePost,NginxWebServer5
914914CAA3FC875F	2019/6/8	10674.630541	175	ComposePost,NginxWebServer5
AB2EC0A5C798E809	2019/6/8	10755.023854	262	ComposePost,NginxWebServer5
C51208AB4CC83EB3	2019/6/8	11153.329817	166	ComposePost,NginxWebServer5
CA672DC97E5DD6D6	2019/6/8	11119.05281	197	ComposePost,NginxWebServer5
013703819EE704E4	2019/6/8	11027.695691	180	ComposePost,NginxWebServer5
1203DF6C017BF6DC	2019/6/8	9472.523541	187	ComposePost,NginxWebServer5
12DF5A71E92B0DDD	2019/6/8	9075.111476	171	ComposePost,NginxWebServer5
236590C559DC69DF	2019/6/8	10498.096384	185	ComposePost,NginxWebServer5
479ABFBB8933860A	2019/6/8	11002.35614	186	ComposePost,NginxWebServer5
5EF912ECF0C61273	2019/6/8	10188.157735	183	ComposePost,NginxWebServer5
664AEC89CA41BEB5	2019/6/8	9497.216001	175	ComposePost,NginxWebServer5
842F8D5781F3B896	2019/6/8	10200.727131	186	ComposePost,NginxWebServer5
9492603F5719FB30	2019/6/8	10891.31312	194	ComposePost,NginxWebServer5
D000067D72E4B239	2019/6/8	8990.666054	167	ComposePost,NginxWebServer5
D4E5390E46797E57	2019/6/8	10511.241623	193	ComposePost,NginxWebServer5

Fig. 2. Example of filtering along the Num Events and Duration dimension to look at one specific set of outliers

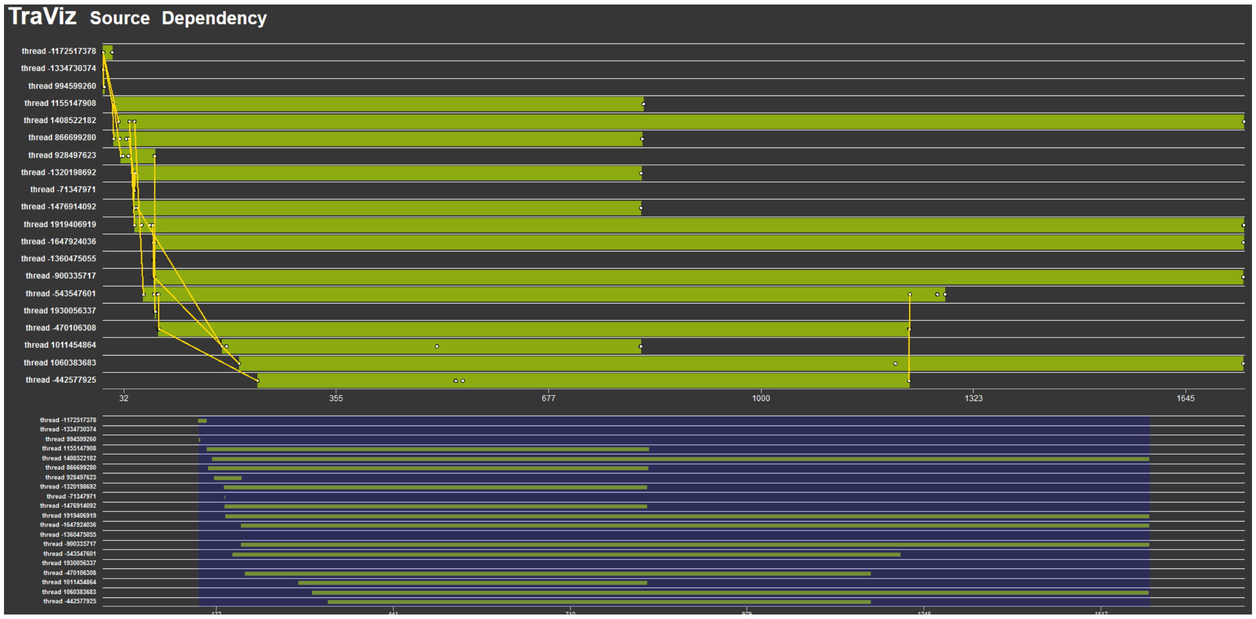
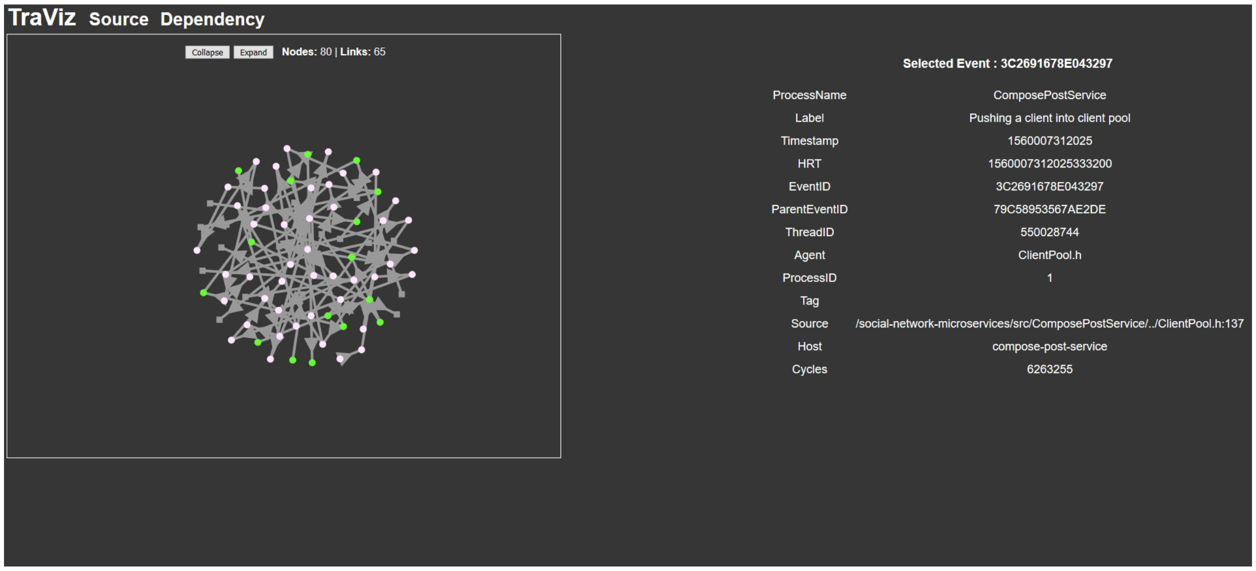


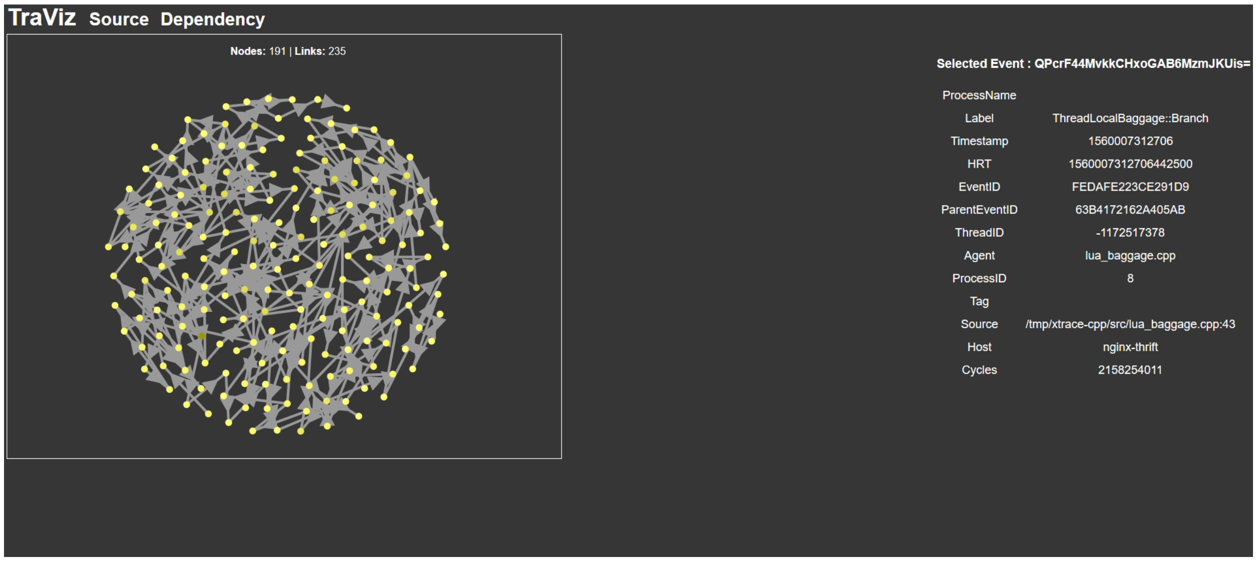
Fig. 3. Swimlane view of trace ID AB2EC0A5C798E809 from the DeathStarBench dataset



Selected Event : 3C2691678E043297

ProcessName	ComposePostService
Label	Pushing a client into client pool
Timestamp	1560007312025
HRT	1560007312025333200
EventID	3C2691678E043297
ParentEventID	79C58953567AE2DE
ThreadID	550028744
Agent	ClientPool.h
ProcessID	1
Tag	
Source	/social-network-microservices/src/ComposePostService/../ClientPool.h:137
Host	compose-post-service
Cycles	6263255

Fig. 4. Comparison of 2 outlier traces from the DeathStarBench dataset



Selected Event : QPcrF44MvkkCHxoGAB6MzmJKUis=

ProcessName	
Label	ThreadLocalBaggage::Branch
Timestamp	1560007312706
HRT	1560007312706442500
EventID	FEDAFE223CE291D9
ParentEventID	63B4172162A405AB
ThreadID	-1172517378
Agent	lua_baggage.cpp
ProcessID	8
Tag	
Source	/tmp/xtrace-cpp/src/lua_baggage.cpp:43
Host	nginx-thrift
Cycles	2158254011

Fig. 5. Aggregation of 20 outlier traces from the DeathStarBench dataset into a single graph

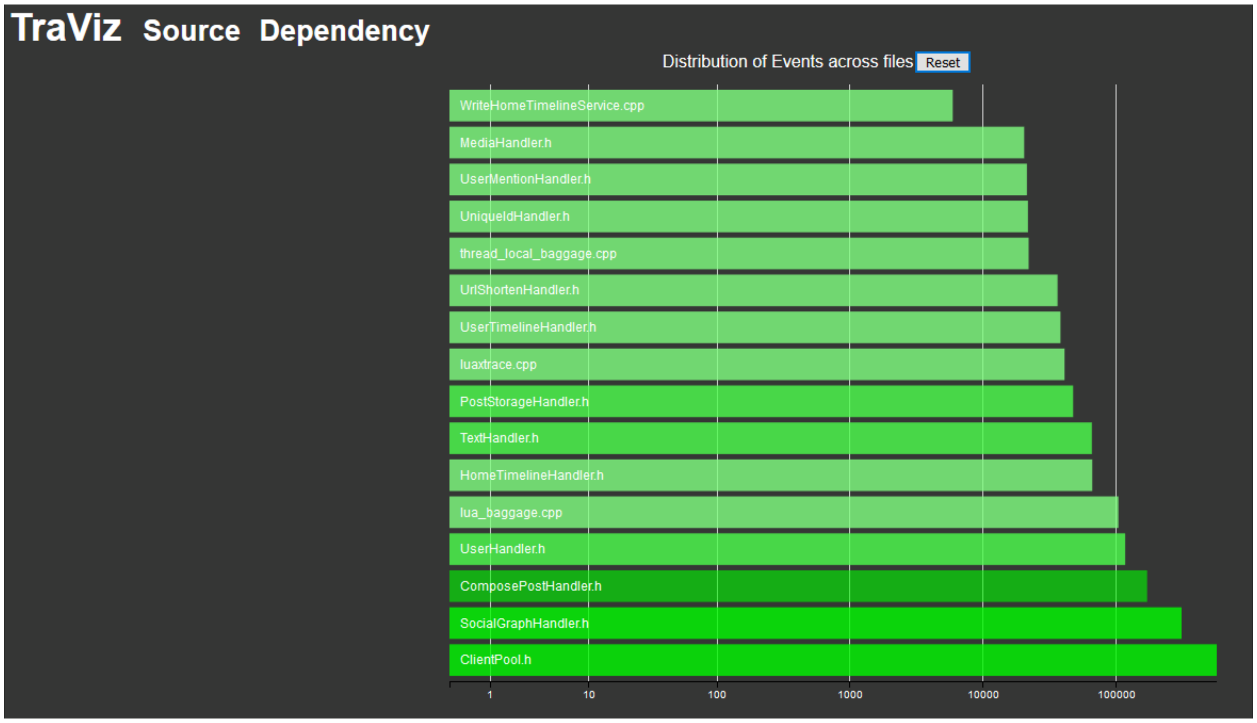


Fig. 6. The distribution of the events across the files for the DeathStarBench dataset, with colour showing the number of lines from the file that produced these events.

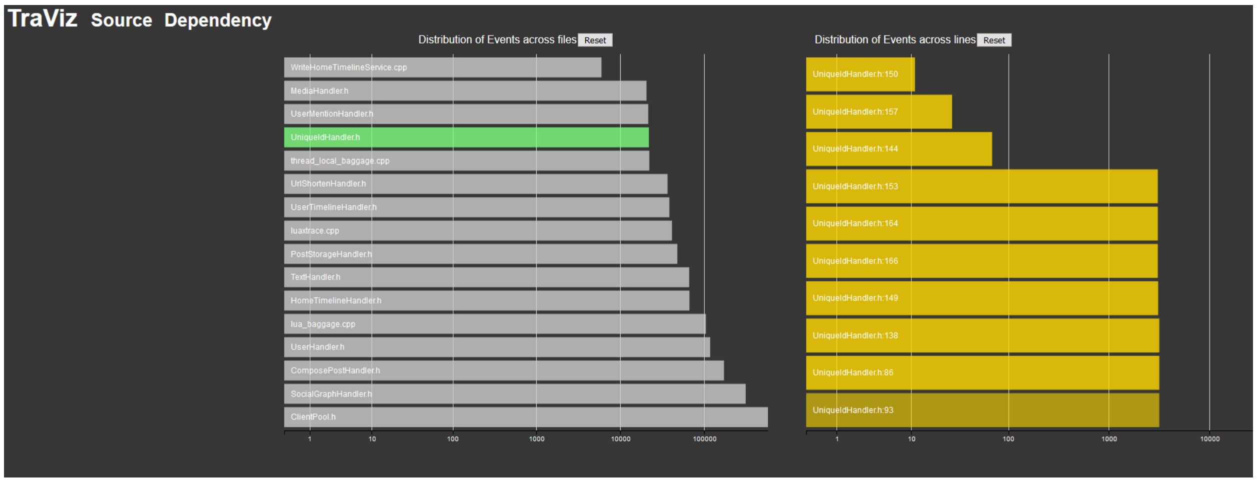
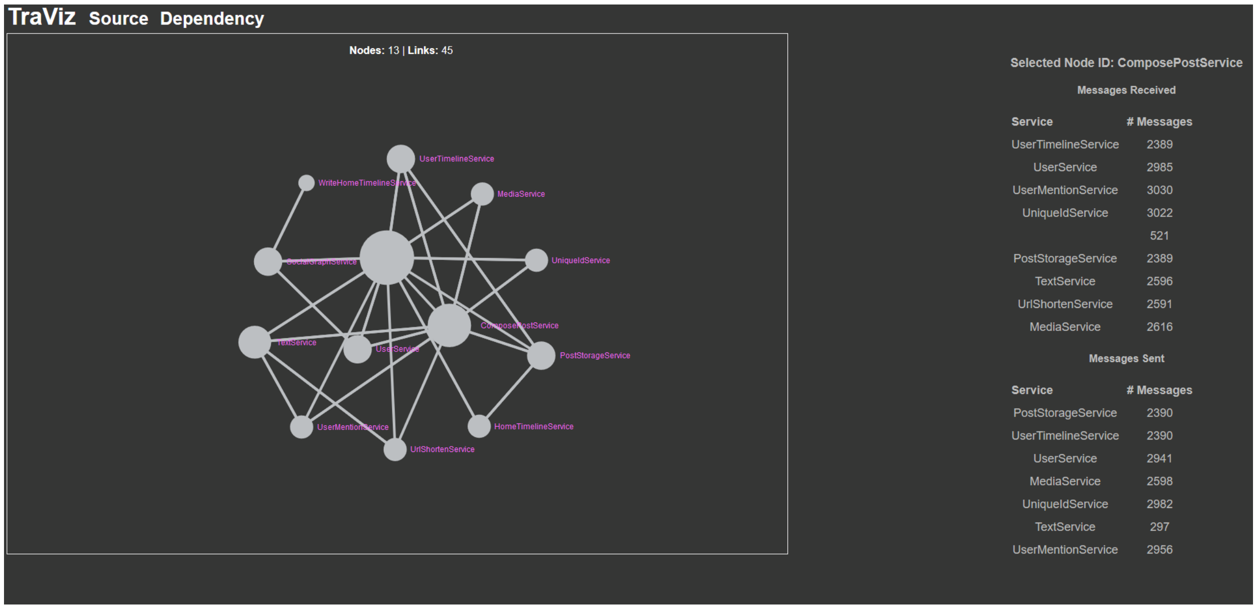


Fig. 7. The distribution of events across the lines for the UniqueIdHandler file



Selected Node ID: ComposePostService

Messages Received

Service	# Messages
UserTimelineService	2389
UserService	2985
UserMentionService	3030
UniqueIdService	3022
	521
PostStorageService	2389
TextService	2596
UrlShortenService	2591
MediaService	2616

Messages Sent

Service	# Messages
PostStorageService	2390
UserTimelineService	2390
UserService	2941
MediaService	2598
UniqueIdService	2982
TextService	297
UserMentionService	2956

Fig. 8. The dependency graph shows how the services in the system interact with each other. The detail view shows the messages received and sent by the nodes running that specific service from the nodes running other services. This figure shows the breakdown for the ComposePostService from the DeathStar Microbenchmark suite.

7.7 Informal User Study

To evaluate the efficacy of TraViz, we conducted a very Informal user study with 2 target users. One of the target users is one of the leading experts in distributed tracing whereas the other user is an engineer at a large internet company where part of their job is to analyze distributed traces. Due to geographical and time constraints, the user study had to be done asynchronously where the target users were given a deployed link of our tool and were asked to provide feedback regarding the user experience.

Both of the target users liked the overview dashboard. Specifically, the leading expert stated ”Incorporating high-level metrics side-by-side with trace search and selection is a good idea and I like the dashboard you came up with”. The source code view also garnered positive reviews as both the users thought this is something that they would like to use with their data. The source code view piqued the interest of the expert user as they thought it had the potential of getting very deep as it was combining the static source code of the system with dynamic information about actual executions. The other user thought the most useful visualization was the individual trace view as it allowed them to view time spent by each thread individually. The expert user thought that the dependency graph was useful but commented that the graph should be directed as that is what users expect to see.

For the trace comparison and aggregation idioms, there was a general sense of confusion across both the users. One of the users was particularly confused as to what each of the nodes meant in the graph - whether the nodes were showing performance information or structural differences or both. The expert user commented that the comparison and aggregation graphs have some underlying ordering which wasn't being visualized. Although, they did acknowledge that getting the ordering right for comparisons and aggregations is a hard task.

Given the feedback from the users, we believe that TraViz is partially successful in its endeavor of being the most usable distributed tracing visual analysis tool.

8 Discussion

We consider TraViz to be partially successful at the time of writing. We believe that TraViz's exploration dashboard is a very good starting point for the users to find potentially outlier traces and then select such traces for further analysis. Additionally, as far as we know, TraViz is the first visualization tool to integrate the static source code information with dynamic performance information captured in the traces. The current source code integration idiom is just a starting point and we believe there are other interesting ways of visualizing and integrating source code information with distributed traces.

We consider TraViz to only be partially successful as we believe that the current graph comparison and aggregation tools are easily usable or understandable. This is primarily due to the fact that our current graph idioms don't have a good layout strategy for laying out the nodes and edges on the screen causing our idioms to suffer from the dreaded ”hairball effect”. Finding a good layout strategy is akin to finding a needle in the haystack but we believe that we will eventually find one that fits our use case.

9 Future Work

Although, we believe that TraViz already provides better visualization idioms for analysing distributed traces, TraViz is far from ready from being deployed in a real world setting.

The main goal for our future work is to improve upon our graph comparison and aggregation idioms so as to remove the "hairball effect". This requires using better layout algorithms for visually laying out graphs on the screen. We have identified potential libraries like cytoscape and d3-dag that provide better layout algorithms but we haven't been able to get those to work yet.

Based on the feedback received from the target users, there are some minor improvements that we would like to make in our existing visualization idioms. Specifically, for the swimlane view of the trace, we would like to further encode the ID of the process as the color of the swimlane for the thread. This would help users identify which process a thread belongs to and understand when the request crosses process boundaries. Additionally, we would like to add a small detail window to the swimlane idiom that shows the details of an event when the corresponding node is clicked on the main visualization. We would also like to switch the dependency graph to be a directed graph as that is what our target users are more accustomed to a directed dependency graph.

Additionally, we want to design idioms for two more tasks. The first task is the comparison between a single trace and an aggregation of traces that would help users understand how a single, anomalous trace is different from a set of normal looking traces. The second task is the comparison between two different aggregations of traces. This would help understand developers to understand the differences in between clusters of traces.

Lastly, we believe that conducting a formal user study or at least an informal user study with more users would help in gauging the efficacy and usability of TraViz as a distributed tracing analysis tool.

10 Conclusion

In this paper we presented TraViz, a tool designed for analyzing the structural and performance attributes of distributed traces. TraViz supports 6 different analysis tasks - detection of outlier traces, single trace performance analysis, source code integration with performance metrics for future code optimizations, comparison of 2 traces to detect differences, aggregation of traces to highlight common structure and find outlier events, and dependency analysis between different services in a distributed system. There has been prior research in visualizing traces but most of it has been focused on performing single trace performance analysis. TraViz extends this by allowing users to explore different traces to find interesting traces and then perform comparison and/or aggregation actions on these traces. As far as we know, TraViz is the first visualization tool that implements a visualization for combining static source code information with dynamic information about the system collected via traces. That being said, we feel that TraViz's graph visualizations are still not useful to developers in its current state and require further refinement before they can be perceived useful.

References

[1] Jaeger - distributed tracing system. `https://www.jaegertracing.io/`. Accessed: 2019-11-04.

[2] Lightstep. `https://lightstep.com/`. Accessed: 2019-11-04.

[3] Shiviz. `https://bestchai.bitbucket.io/shiviz/`. Accessed: 2019-11-04.

[4] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, pages 20–20. USENIX Association, 2007.

[5] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18. ACM, 2019.

[6] M. Klein. Distributed tracing costs are not justified - a thread. `https://twitter.com/mattklein123/status/1049813546077323264`. Accessed: 2019-12-12.

[7] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[8] J. Mace. X-trace server. `https://gitlab.mpi-sws.org/cld/tracing/x-trace-server`. Accessed: 2019-12-12.

[9] N. Shervashidze, P. Schweitzer, E. J. v. Leeuwen, K. Mehlhorn, and K. M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(Sep):2539–2561, 2011.

[10] C. Sridharan. Distributed tracing — we've been doing it wrong. Accessed: 2019-12-12.